David Kirby

Homework Assignment 2

ECE438

Due 3/15/2020

1. **(10 points)** A program that you would like to run on a MIPS based microprocessor has a dynamic instruction count of $2.0 * 10^6$ and can be divided up as follows:

Instruction	**Rate of Occurence**
beq/bne	10%
j	1%
jal	2%
jr	2%
addu	22%
lui	2%
sll	6%
slt/slti	9%
and/andi	6%
lw	21%
sw	9%
lb	8%
sb	2%

   (a) Assuming the average CPI for the MIPS machine is 1.3 for loads and stores, 1.2 for control flow instructions, and 1.1 for ALU instructions, what is the overall average CPI for the instruction mix shown above?

   (b) Assuming the machine has a clock rate of 2.5GHz, calculate the execution time of this program.

   (c) If you design another machine that has a 3.5GHz clock rate with an average CPI of 1.2 for loads and stores, 1.1 for control flow instructions and 0.9 for ALU instructions, how much faster would the new machine be?

2. **(10 points)** For each of the MIPS instructions below, show the value of the *opcode*, *rs*, and *rt*. For R-type instructions, provide the value of *rd*, and for I-type instructions, provide the *immediate* value.

```
add $t4, $t5, $t6
lw $t2, 8($t4)
lw $t3, 12($t4)
xor $t6, $t2, $t3
addi $t4, $t4, 8
sw $t6, 24($t4)
```

3. **(5 points)** With the following binary value (machine code), determine the exact assembly instruction it represents. What type of instruction is it (R-type, I-type, or J-type)?

$$0000\ 0010\ 0001\ 0000\ 1000\ 0000\ 0010\ 0000_2$$

4. **(5 points)** With the following assembly instruction, generate the machine code:

```
sw $t0, -16($sp)
```

5. **(10 points)** For the following problems, consider the binary address below:

$$0010\ 0000\ 0000\ 0001\ 0100\ 1001\ 0010\ 0100_2$$

   (a) Write the MIPS assembly code that stores the 32-bit address above into the $t1 register.

   (b) If the current value of the PC is `0x00000000`, can you use a single jump (J-type) instruction to get to the address show above in binary?

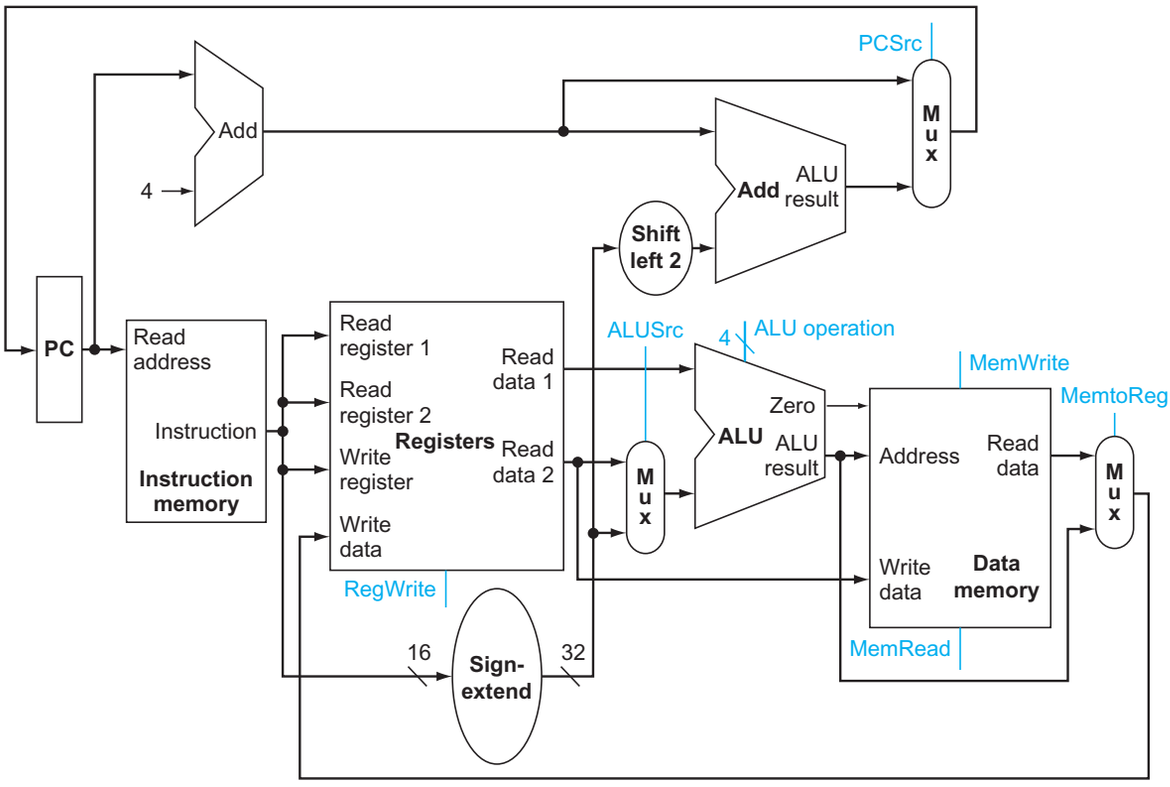
   (c) What about if the current value of the PC is `0x00000600`?

   (d) What about if the current value of the PC is `0x1FFFF000`?

   (e) If you *cannot* use a single J-type instruction to get to the address above, how would you go about jumping to that address?

6. **(20 points)** Consider the single-cycle datapath shown below with the following component latencies:

I-mem	Add	Mux	ALU	Reg Read	Reg Write	D-Mem	Sign-Extend	Shift-Left-2
200ps	90ps	15ps	100ps	90ps	40ps	200ps	15ps	10ps



(a) What is the maximum achievable clock rate for the datapath above? Ignore the effects of the control unit.
Hint: Mark the arrival time of valid data on each of the components in the figure above.

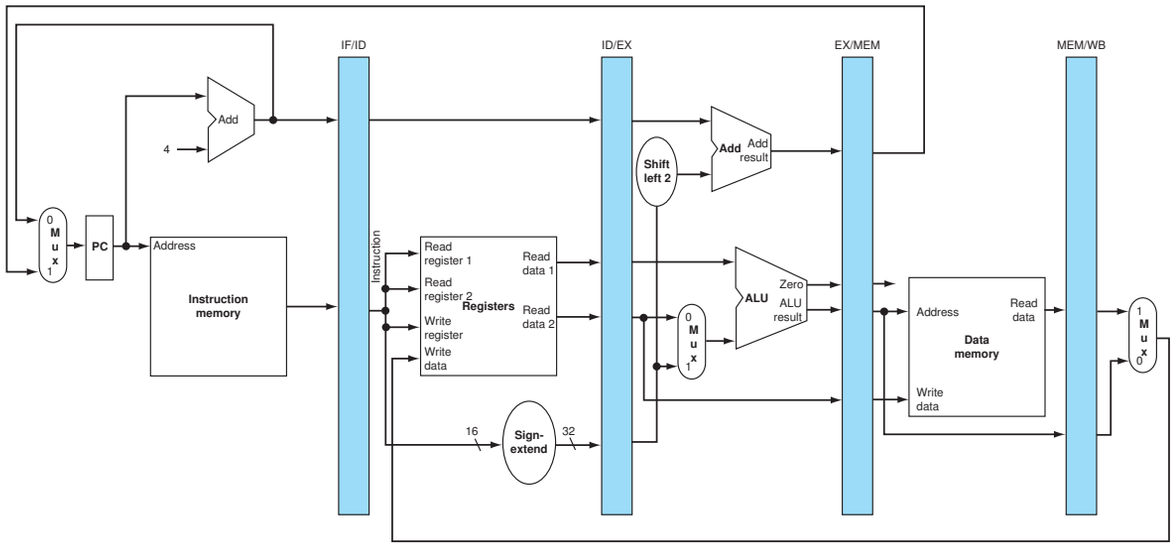
(b) What is the latency of a load instruction in this datapath?

(c) What is the latency of a store instruction?

(d) What is the latency of an ALU instruction?

(e) What is the CPI of this design?

(f) Consider the pipelined datapath shown below. Ignoring the effects of the control unit and flip-flop delays, what is the maximum achievable clock rate?



(g) What would be the latency of a load instruction in the pipelined datapath?

(h) What would be the latency of an ALU instruction?

(i) Assuming a long running program with no pipeline hazards, what is the instruction throughput of the above pipelined processor? What is the speedup compared to the single-cycle machine?

(j) What is the maximum achievable clock rate of the pipelined datapath above if we account for a 20ps flip-flop delay.

(k) Why is the single-cycle design impractical?

(l) Compare the speedup you calculated in Problem 6i with that of the ideal speedup for a 5-stage pipeline. What are some things that reduce the speedup of pipelining?

7. **(10 points)** Consider the code below:

```
add $t1, $t2, $t3
lw $t4, 0($t1)
lw $t5, 4($t1)
lw $t6, 8($t1)
xor $t7, $t4, $t5
xor $t7, $t7, $t6
sw $t7, 24($t1)
addi $t1, $t1, 8
```

(a) Identify the data dependencies in the code above.

(b) Assuming a 5-stage pipelined processor without forwarding or hardware interlocking (stalling), rewrite the code above with `nops` to ensure correct execution. Also assume register write happens in the beginning of the clock cycle, and read happens at the end.

(c) Assuming a 5-stage pipelined processor with full data forwarding but no hardware interlocking (stalling), rewrite the code with `nops` to ensure correct execution.

(d) Reorder the code from part (c) to eliminate as many `nops` as possible.

8. **(15 points)** For the following problem, assume a 5-stage pipelined processor with a branch delay slot and branch resolution in the Execute stage. Also assume the pipeline has full forwarding and hardware interlocking. Consider the code below:

```
    lw $t2, 0($t1)
label1:
    beq $t2, $t0, label2 #not taken once, then taken
    lw $t3, 0($t2)
    beq $t3, $t0, label1 #taken
    add $t1, $t3, $t1
label2:
    sw $t1, 0($t2)
```

   (a) Draw the pipeline execution diagram for the above code when an "assume not taken" branching scheme is used. Assume the code above has already been arranged to fill the branch delay slots.

   (b) How many clock cycles are required to execute the code above when an "assume not taken" branching scheme is used?

   (c) Redraw the pipeline execution diagram assuming a 100% correct branch predictor.

   (d) What speedup does the branch predictor provide over the "assume not taken" scheme?

   (e) If the branch decision was moved to the Decode stage, how many clock cycles would be required? Draw the pipeline execution diagram.

   (f) Assuming `label1` is at address `0x20000010`, provide the machine code for both branches from the assembly code above.

9. **(10 points)** In the following problem, assume we are starting with a *single-cycle* design with the following latencies:

I-mem	Add	Mux	ALU	Reg Read	Reg Write	D-Mem	Sign-Extend	Shift-Left-2
200ps	90ps	15ps	100ps	90ps	40ps	200ps	15ps	10ps

Consider the addition of a multiplier to the ALU that will add 300ps to the latency of the ALU but will reduce the dynamic instruction count by 20% due to the fact that the multiply instruction will no longer have to be emulated in software.

   (a) Compare the cycle time and clock rate of the *single-cycle* design with and without the multiplier.

   (b) Calculate the speedup achieved by adding the multiplier.

   (c) Consider the addition of the multiplier to our pipelined design in Problem 6f. If the multiplier adds an additional 300ps of latency to the ALU, what is the clock rate of the pipelined processor with the multiplier?

   (d) Does the multiplier still provide a speedup?

   (e) In keeping with our pipelined theme, we decide to pull the multiplier out of the ALU and break it up into two equal stages. Assuming we can break it up such that it requires 150ps per stage, recalculate the clock rate of our pipeline processor with the pipelined multiplier.

   (f) Ignoring the additional data hazards a pipelined multiplier might cause, recalculate the speedup of our pipelined processor with the pipelined multiplier compared to the pipelined processor without a multiplier.

1. **(10 points)** A program that you would like to run on a MIPS based microprocessor has a dynamic instruction count of $2.0 * 10^6$ and can be divided up as follows:

2 million instr.

Instruction	Rate of Occurence	
beq/bne	10%	= 0.12
j	1%	= 0.012
jal	2%	= 0.024
jr	2%	= 0.024
addu	22%	= 0.242
lui	2%	= 0.022
sll	6%	= 0.066
slt/slti	9%	= 0.099
and/andi	6%	= 0.066
lw	21%	= 0.273
sw	9%	= 0.117
lb	8%	= 0.104
sb	2%	= 0.026

control flow = 1.2 → 0.180

ALU = 1.1 → 0.495 → Σ = 1.195

loads & stores = 1.3 → 0.520

(a) Assuming the average CPI for the MIPS machine is 1.3 for loads and stores, 1.2 for control flow instructions, and 1.1 for ALU instructions, what is the overall average CPI for the instruction mix shown above?

(b) Assuming the machine has a clock rate of 2.5GHz, calculate the execution time of this program.

(c) If you design another machine that has a 3.5GHz clock rate with an average CPI of 1.2 for loads and stores, 1.1 for control flow instructions and 0.9 for ALU instructions, how much faster would the new machine be?

(a) The overall average CPI for the instruction mix is **1.195**

(b) $time_{exec} = CPI \times \frac{1}{clk} \times \#\,instr. = \frac{(1.195)(2e^6)}{2.5\,GHz} =$ **956 µs**

(c)

Instruction	Rate of Occurence	
beq/bne	10%	= 0.11
j	1%	= 0.011
jal	2%	= 0.022
jr	2%	= 0.022
addu	22%	= 0.198
lui	2%	= 0.018
sll	6%	= 0.054
slt/slti	9%	= 0.081
and/andi	6%	= 0.054
lw	21%	= 0.252
sw	9%	= 0.108
lb	8%	= 0.096
sb	2%	= 0.024

control flow = 1.1 → 0.165

ALU = 0.9 → 0.405 → Σ = 1.05

loads & stores = 1.2 → 0.48

The new machine would be **1.138×** faster.

2. **(10 points)** For each of the MIPS instructions below, show the value of the *opcode*, *rs*, and *rt*. For R-type instructions, provide the value of *rd*, and for I-type instructions, provide the *immediate* value.

	type	opcode (bin)	(dec)	rs	rt	rd	immed.
add $t4, $t5, $t6	R-type	000000	0	13	14	12	—
lw $t2, 8($t4)	I-type	100011	35	12	10	—	8
lw $t3, 12($t4)	I-type	100011	35	12	11	—	12
xor $t6, $t2, $t3	R-type	000000	0	10	11	14	—
addi $t4, $t4, 8	I-type	001000	8	12	12	—	8
sw $t6, 24($t4)	I-type	101011	43	14	12	—	24

Registers		
0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7
8	9	10	11	12	13	14	15

3. **(5 points)** With the following binary value (machine code), determine the exact assembly instruction it represents. What type of instruction is it (R-type, I-type, or J-type)?

31:26 25:21 20:16 15:11 10:6 5:0

$0000\ 0010\ 0001\ 0000\ 1000\ 0000\ 0010\ 0000_2$

0 | 16 | 16 | 16 | 0 | 32

R-type | Rs | Rt | Rd

↓

add

add $s0, $s0, $s0

R-type

$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7
16	17	18	19	20	21	22	23

4. **(5 points)** With the following assembly instruction, generate the machine code:

```
sw $t0, -16($sp)
```

	rs	rt		instr	
31:26	25:21	20:16	15:11	10:6	5:0
sw	$sp = 29	$t0 = 8			-16
1010.11	11.101	0.1000.	1111.1	111.11	11.0000_2

I-type
sign extended

5. **(10 points)** For the following problems, consider the binary address below:

$$0010\ 0000\ 0000\ 0001\ 0100\ 1001\ 0010\ 0100_2$$

(a) Write the MIPS assembly code that stores the 32-bit address above into the $t1 register.

(b) If the current value of the PC is `0x00000000`, can you use a single jump (J-type) instruction to get to the address show above in binary?

(c) What about if the current value of the PC is `0x00000600`?

(d) What about if the current value of the PC is `0x1FFFF000`?

(e) If you *cannot* use a single J-type instruction to get to the address above, how would you go about jumping to that address?

$0010.00\ 00.000\ 0.0001.\quad 0100.1\ 001.00\ 10.0100_2$

$8193_{10} = 0x2001_{16}$ $\qquad$ $18724_{10} = 0x4924_{16}$

(a) lui $t1, 0x2001

ori $t1, $t1, 0x4924

(b) According to the MIPs manual, jumps can branch within 256MB shifted left 2 bits.

$256 \times 1024^2 - 4$

$= 268,435,452$

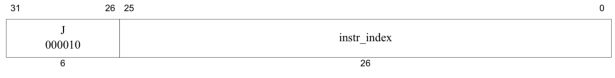
$= 0x0FFFFFFC$

$0x0FFFFFFC - 0x20014924$

$= -10014928$

It is not possible to get to our address in a single jump.

J — Jump



Format: `J target` **MIPS32**

Purpose: Jump

To branch within the current 256 MB-aligned region.

Description:

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

Restrictions:

Control Transfer Instructions (CTIs) should not be placed in branch delay slots or Release 6 forbidden slots. CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE.

(c) $0x0FFFFFFC - 0x20014924 + 0x600$

$= -10014328$

It is still not possible to get to our address in a single jump.

(d) $0x0FFFFFFC - 0x20014924 + 0x1FFFF000$

$= 0xFFEA6DB$

Yes, it is possible to get to our address in a single jump.

(e) We could use PC-relative addressing and jump registers.

6. **(20 points)** Consider the single-cycle datapath shown below with the following component latencies:

I-mem	Add	Mux	ALU	Reg Read	Reg Write	D-Mem	Sign-Extend	Shift-Left-2
200ps	90ps	15ps	100ps	90ps	40ps	200ps	15ps	10ps

(a) What is the maximum achievable clock rate for the datapath above? Ignore the effects of the control unit.
Hint: Mark the arrival time of valid data on each of the components in the figure above.

(b) What is the latency of a load instruction in this datapath?

(c) What is the latency of a store instruction?

(d) What is the latency of an ALU instruction?

(e) What is the CPI of this design?

(f) Consider the pipelined datapath shown below. Ignoring the effects of the control unit and flip-flop delays, what is the maximum achievable clock rate?

(g) What would be the latency of a load instruction in the pipelined datapath?

(h) What would be the latency of an ALU instruction?

(i) Assuming a long running program with no pipeline hazards, what is the instruction throughput of the above pipelined processor? What is the speedup compared to the single-cycle machine?

(j) What is the maximum achievable clock rate of the pipelined datapath above if we account for a 20ps flip-flop delay.

(k) Why is the single-cycle design impractical?

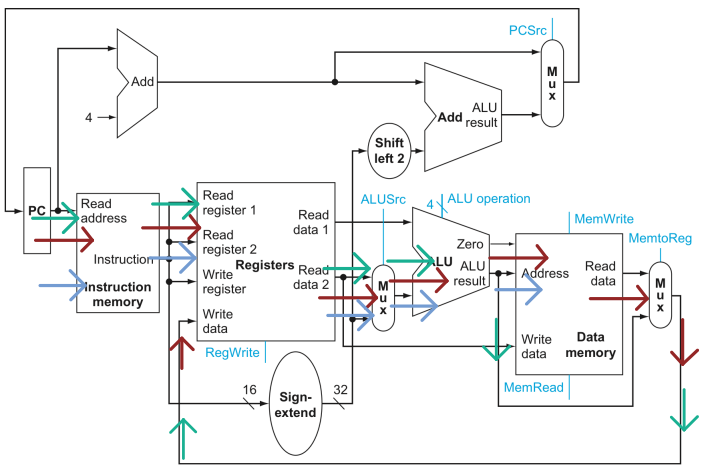
(l) Compare the speedup you calculated in Problem 6i with that of the ideal speedup for a 5-stage pipeline. What are some things that reduce the speedup of pipelining?

(a) 200 ps

(b) I-mem + Reg read + MUX + ALU + D-Mem + Mux + Reg Write

200 + 90 + 15 + 100 + 200 + 15 + 40

$Latency_{lw}$ = 660 ps



(c) I-mem + Reg read + MUX + ALU + D-mem

200 + 90 + 15 + 100 + 200

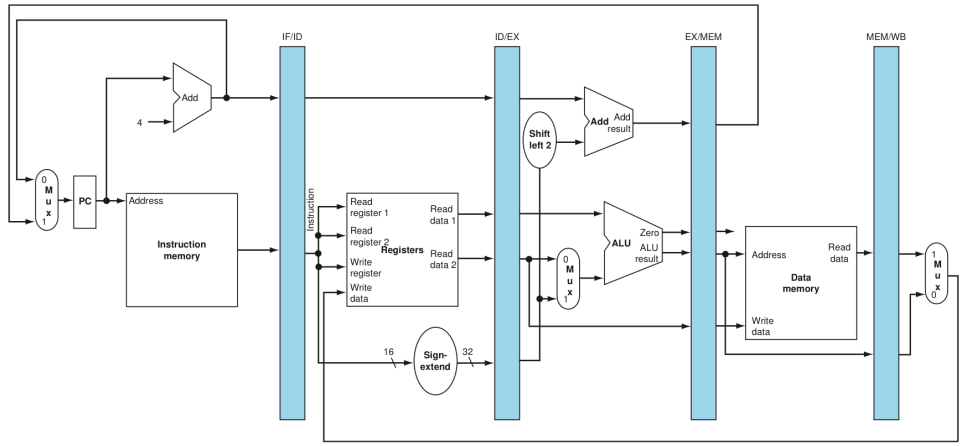
$Latency_{sw}$ = 605 ps

(d) I-mem + Reg read + MUX + ALU + MUX + Reg write

200 + 90 + 15 + 100 + 15 + 40

$Latency_{ALU}$ = 460 ps

(e) The critical path is LW at 7 cycles — ?


IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
16
Sign-extend
32

7. **(10 points)** Consider the code below:

```
add $t1, $t2, $t3
lw $t4, 0($t1)
lw $t5, 4($t1)
lw $t6, 8($t1)
xor $t7, $t4, $t5
xor $t7, $t7, $t6
sw $t7, 24($t1)
addi $t1, $t1, 8
```

(a) Identify the data dependencies in the code above.

(b) Assuming a 5-stage pipelined processor without forwarding or hardware interlocking (stalling), rewrite the code above with `nops` to ensure correct execution. Also assume register write happens in the beginning of the clock cycle, and read happens at the end.

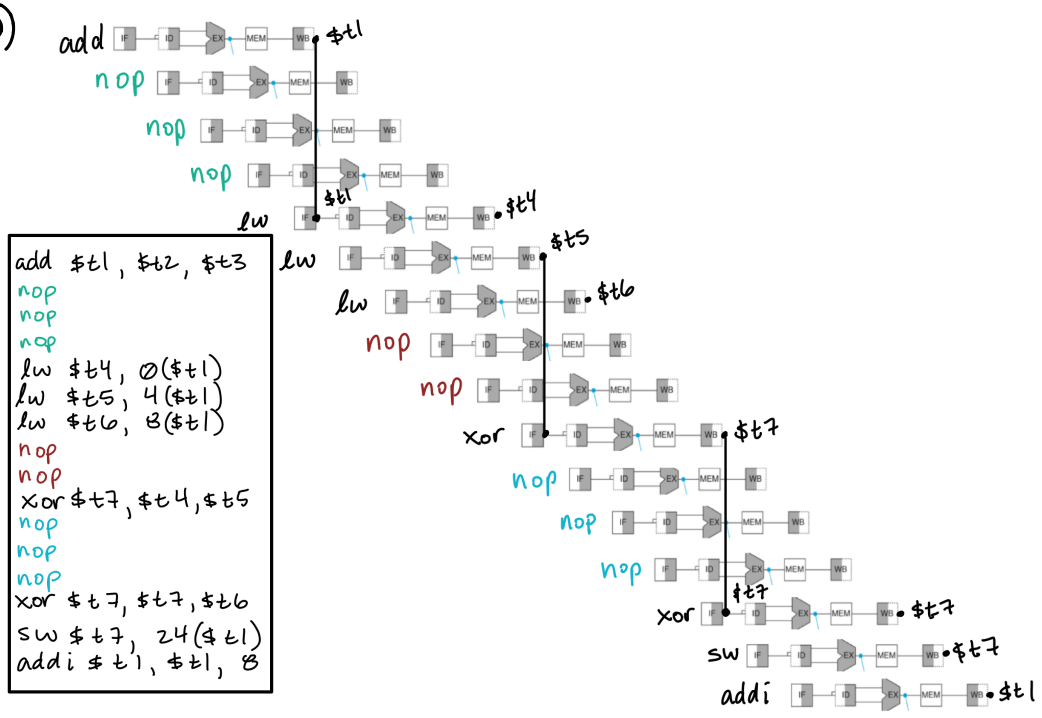
(c) Assuming a 5-stage pipelined processor with full data forwarding but no hardware interlocking (stalling), rewrite the code with `nops` to ensure correct execution.

(d) Reorder the code from part (c) to eliminate as many `nops` as possible.

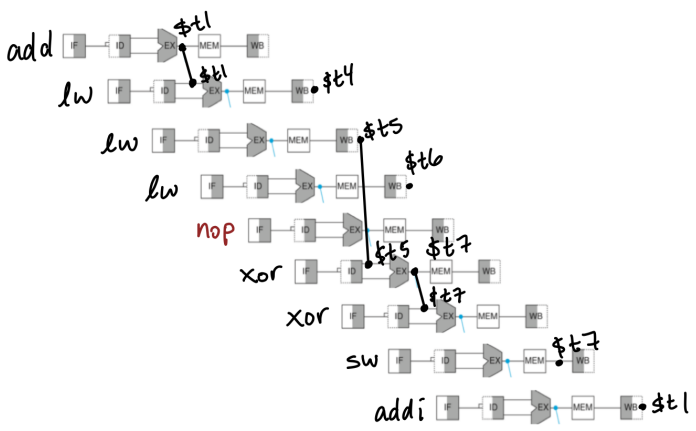
(a)

```
add $t1, $t2, $t3
lw $t4, 0($t1)
lw $t5, 4($t1)
lw $t6, 8($t1)
xor $t7, $t4, $t5
xor $t7, $t7, $t6
sw $t7, 24($t1)
addi $t1, $t1, 8
```

(b)



(c)



(d)

```
lw $t4, 0($t1)
lw $t5, 4($t1)
lw $t6, 8($t1)
add $t1, $t2, $t3
xor $t7, $t4, $t5
xor $t7, $t7, $t6
sw $t7, 24($t1)
addi $t1, $t1, 8
```

8. **(15 points)** For the following problem, assume a 5-stage pipelined processor with a branch delay slot and branch resolution in the Execute stage. Also assume the pipeline has full forwarding and hardware interlocking. Consider the code below:

```
    lw $t2, 0($t1)
label1:
    beq $t2, $t0, label2 #not taken once, then taken
    lw $t3, 0($t2)
    beq $t3, $t0, label1 #taken
    add $t1, $t3, $t1
label2:
    sw $t1, 0($t2)
```

(a) Draw the pipeline execution diagram for the above code when an "assume not taken" branching scheme is used. Assume the code above has already been arranged to fill the branch delay slots.

(b) How many clock cycles are required to execute the code above when an "assume not taken" branching scheme is used?

(c) Redraw the pipeline execution diagram assuming a 100% correct branch predictor.

(d) What speedup does the branch predictor provide over the "assume not taken" scheme?

(e) If the branch decision was moved to the Decode stage, how many clock cycles would be required? Draw the pipeline execution diagram.

(f) Assuming `label1` is at address `0x20000010`, provide the machine code for both branches from the assembly code above.

(a) If "assume not taken", both beq will essentially be ignored.

	1	2	3	4	5	6	7	8	9	10
lw	IF	ID	Ex	Mem	WB					
beq		IF	ID	Ex	Mem	WB				
lw			IF	ID	Ex	Mem	WB			
beq				IF	ID	Ex	Mem	WB		
add					IF	ID	Ex	Mem	WB	
sw						IF	ID	Ex	Mem	WB

(b) It would take 10 cycles.

(c)

	1	2	3	4	5	6	7	8	9	10
lw	IF	ID	Ex	Mem	WB					
beq		IF	ID	Ex	Mem	WB				
lw			IF	ID	Ex	Mem	WB			
sw				IF	ID	Ex	Mem	WB		

(d) $\frac{3}{10}$ = 0.3x speedup

(e) not sure what this means

(f) again, not sure what this means

9. **(10 points)** In the following problem, assume we are starting with a *single-cycle* design with the following latencies: 400ps

I-mem	Add	Mux	ALU	Reg Read	Reg Write	D-Mem	Sign-Extend	Shift-Left-2
200ps	90ps	15ps	100ps	90ps	40ps	200ps	15ps	10ps

Consider the addition of a multiplier to the ALU that will add 300ps to the latency of the ALU but will reduce the dynamic instruction count by 20% due to the fact that the multiply instruction will no longer have to be emulated in software.

(a) Compare the cycle time and clock rate of the *single-cycle* design with and without the multiplier.

(b) Calculate the speedup achieved by adding the multiplier.

(c) Consider the addition of the multiplier to our pipelined design in Problem 6f. If the multiplier adds an additional 300ps of latency to the ALU, what is the clock rate of the pipelined processor with the multiplier?

(d) Does the multiplier still provide a speedup?

(e) In keeping with our pipelined theme, we decide to pull the multiplier out of the ALU and break it up into two equal stages. Assuming we can break it up such that it requires 150ps per stage, recalculate the clock rate of our pipeline processor with the pipelined multiplier.

(f) Ignoring the additional data hazards a pipelined multiplier might cause, recalculate the speedup of our pipelined processor with the pipelined multiplier compared to the pipelined processor without a multiplier.

(a) without multiplier = 760ps ; 1.316 GHz